2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Compiler Support for Optimizing Memory Bank-Level Parallelism

Wei Ding, Diana Guttman, and Mahmut Kandemir
The Pennsylvania State University, University Park, Pennsylvania, USA
{wzd109, drg217, kandemir}@cse.psu.edu

***Abstract*—Many prior compiler-based optimization schemes focused exclusively on cache data locality. However, cache locality is only one part of the overall performance of applications running on emerging multicores or manycores. For example, memory stalls could constitute a very large fraction of execution time even in cache-optimized codes, and one of the main reasons for this is lack of memory-level parallelism. Motivated by this, we propose a compiler-based Bank-Level Parallelism (BLP) optimization scheme that uses loop tile scheduling. More specifically, we first use Cache Miss Equations to predict where the last-level cache miss will happen in each tile, and then identify the set of memory banks that will be accessed in each tile. Using this information, two tile scheduling algorithms are proposed to maximize BLP, each targeting a different scenario. We further discuss how our compiler-based scheme can be enhanced to consider memory controller-level parallelism and row-buffer locality. Our experimental evaluation using 11 multithreaded applications shows that the proposed BLP optimization can improve average BLP by 17.1% on average, resulting in a 9.2% reduction in average memory access latency. Furthermore, considering memory controller-level parallelism and row-buffer locality (in addition to BLP) takes our average improvement in memory access latency to 22.2%.**

I. Introduction

Due to the importance of caches in shaping performance of both sequential and parallel applications, today, we have a very rich literature on data locality [44], [16], [42], [22], [36], [43]. However, cache locality is only one part of the big picture in emerging multicore/manycore systems, as a lot of data-intensive applications (e.g., scientific programs) incur a significant number of last-level cache misses, making it important to consider "post-miss" scenarios as well. Existing compilers are tuned exclusively to minimize the number of cache misses and do not worry about what happens to the misses. This is unfortunate because last-level cache misses present both challenges and opportunities as far as overall application performance is concerned. For example, several studies [19], [20] indicate that memory stalls constitute a very large fraction of execution time of even cache-optimized application codes, and others [38], [19], [20] show that the distribution of cache misses is not uniform, i.e., some misses can take significantly longer latencies than others, depending on memory queue lengths, row-buffer hit rates, bank-conflicts, etc. Consequently, exposing cache misses to compilers can potentially bring significant data access optimization opportunities, and also complement pure hardware-based cache miss optimization schemes. A few recent compiler-based efforts along this direction consider mainly row-buffer optimizations using data layout restructuring [13].

One of the critical parameters that shape the performance of last-level cache misses is "bank-level parallelism" (BLP), which can be defined as the *number of concurrent accesses to different memory banks* in the system. BLP can change from one workload/application to another or even across different phases of the same workload/application and, as documented by previous research [23], [21], [31], a high BLP usually correlates very well with low memory access latency. Figure 1(a) gives the bank-level parallelism (BLP) and cycles per load instruction (CPL) numbers for our original benchmarks (averaged over the entire execution), when they are tiled (i.e., iteration spaces of loops are divided into small chunks) and tiles are scheduled without considering BLP.[1] One can observe from this plot that the resulting BLP values are not high, considering the fact that our default memory system has 64 banks. In addition, we see that a low BLP usually corresponds to a high CPL and vice versa. As a result, improving BLP can help us reduce the memory cycles, and eventually, improve overall application performance. Figures 1(b), (c) and (d) give the variations of BLP over the course of execution for three of our applications. One can identify from these plots the different execution phases (roughly, corresponding to different loop nests) from the perspective of BLP. For example, in *gafort*, we have a phase between epochs 24 and 36, and another phase between epochs 38 and 45. Our proposed approach in this work tries to optimize BLP in each major phase of an application.

While there are several papers [17], [21], [29] that propose hardware-based techniques that quantify BLP and/or optimize it, we are *not* aware of any compiler-based effort on optimizing BLP in multithreaded applications that targets multicore and manycore systems. Compared to hardware-based schemes, a compiler-based approach can have a limited scope (namely, compiler-analyzable codes). However, in cases where it is applicable, it can bring much better improvements than hardware schemes as the compiler can analyze future data access patterns and restructure code to maximize BLP. Furthermore, in many cases, a compiler-based scheme can also complement the hardware schemes. Motivated by this, in this paper, we make the following **contributions**:

- We propose a compiler-based BLP optimization scheme

[1] Our experimental setup is given in Section VIII.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.34

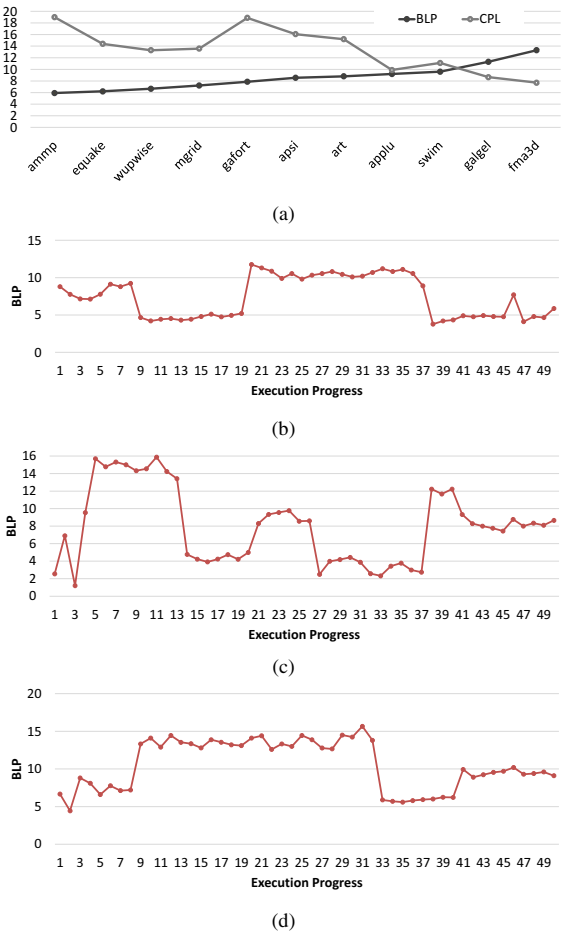

Fig. 1: (a) BLP and cycles-per-load in our applications. (b-d) BLP variation over time for *gafort*, *apsi*, and *swim*.

that employs iteration space tile scheduling. Our scheme first uses Cache Miss Equations [14] to predict where the last-level cache miss will happen in each tile, and then identifies the memory banks that will be accessed in each tile. Using this information, two tile scheduling algorithms are proposed to maximize BLP in different cases.

- We discuss how our scheme can be enhanced to consider memory controller-level parallelism (CLP) and row-buffer locality (RBL), in addition to BLP. Two enhanced versions built upon our BLP optimization algorithm are proposed by adding various constraints to our original formulation of the problem.
- We present a comprehensive experimental evaluation of our schemes using 11 multithreaded applications. Our results show that the proposed optimization can improve BLP by 17.1% on average, resulting in 9.2% reduction in average memory access latency. Furthermore, considering memory controller-level parallelism and row-buffer locality (in addition to BLP) takes our average improvement in memory access latency to 22.2%.

II. Background

A. Loop Tiling and Scheduling

In this work, we focus on loop/data-intensive programs containing affine loop nests, in which loop bounds and array references are affine functions of enclosing loop indices and other global parameters (e.g., input size). Loop tiling (also called iteration space tiling or loop blocking) is an optimization targeting this type of loop that partitions a loop's iteration space into small tiles or blocks. The *iteration space* of an n-level loop nest can be viewed as an n-dimensional polyhedron bounded by the loop bounds. Figure 2(a) shows an example code before and after tiling, and Figure 2(b) illustrates the corresponding iteration space. Each iteration (each point in this polyhedron) can be expressed by an *iteration vector* $\vec{i} = (i_1 i_2 \cdots i_n)^T$, where $i_1, i_2, \cdots, i_n$ are the loop iterators. An array reference $\vec{r}$ within this loop nest can be expressed as an affine function of the iteration vector, which can be written as

$$\vec{r} = A\vec{i} + \vec{o}, \tag{1}$$

where A and $\vec{o}$ are a constant matrix and a constant vector, respectively. For example, the reference $X[i_1][3i_2+1]$ in a two-level loop nest (with loop iterators i_1 and i_2) can be expressed as: $\vec{r} = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \cdot \vec{i} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, where $\vec{i} = (i_1 \;\; i_2)^T$.

There is a rich literature on tiling, focusing on both loop-level parallelism and data locality aspects [44], [16], [42], [22], [36], [43]. When tiles are used as the "units" of parallel execution, many existing compiler-directed loop optimization techniques [10], [26], [49], [36] make use of a *Data Dependence Graph* (DDG) (in some form) to guide the tile scheduling decisions. Figure 2(c) illustrates the DDG that corresponds to the tiled code shown in Figure 2(a). This graph can be constructed by the data-dependence analysis in the compiler and gives execution constraints among different tiles. More specifically, each node in a DDG represents a tile, and an edge from one node to another indicates a data dependence between corresponding tiles. For example, in Figure 2(c), tiles T_4, T_5, T_6 and T_7 cannot be executed until the execution of iterations in T_0, T_1, T_2 and T_3 is finished, due to the data dependence among them (captured by arrows). Therefore, $\{T_4, T_5, T_6, T_7\}$ and $\{T_0, T_1, T_2, T_3\}$ form two *dependence-free sets*, and in each set, the tiles can be scheduled at the same logical time slot (at different cores) to maximize application (loop) level parallelism. In loop/data-intensive programs, the loop bounds are usually quite large. Therefore, the number of tiles in most dependence-free sets (after tiling) is also large. This provides us with the flexibility to select the most suitable set of tiles to schedule at each scheduling slot. In this paper, we take advantage of this flexibility, and schedule tiles in a BLP-aware fashion. Note that we are not proposing a tile shape/size selection strategy. Tile shape/size is usually dictated by cache locality and (loop-level) parallelism concerns, which are beyond our paper. Our proposed tile scheduling strategy works with any tile shape/size.

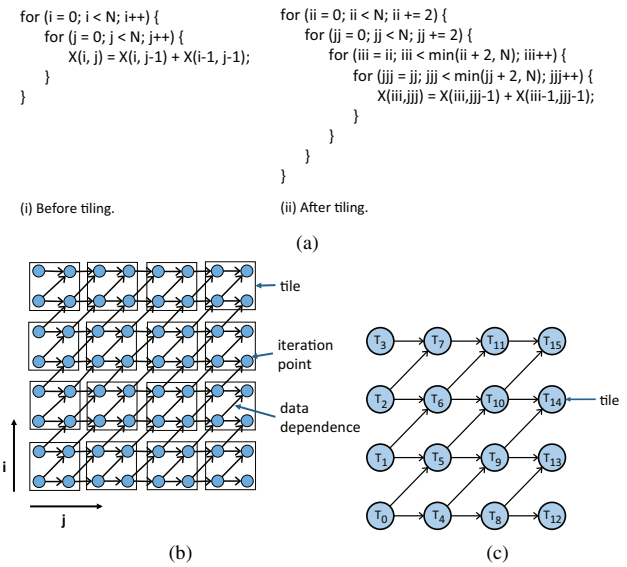

Fig. 2: (a) An example of loop tiling. (b) The corresponding iteration space. (c) The corresponding data dependence graph (DDG). Arrows capture the data dependences among tiles.

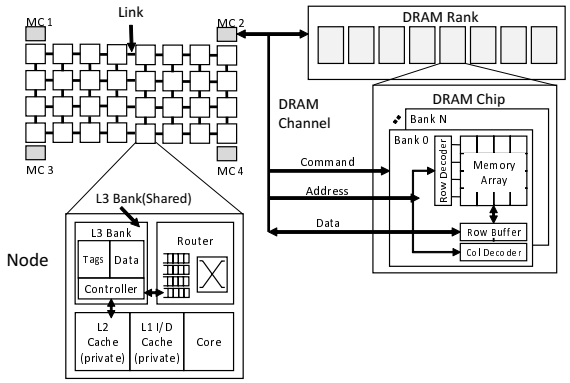

Fig. 3: High-level organization of our target NOC-based manycore and the main memory (DRAM).

B. Bank-Level Parallelism (BLP)

Modern DRAM chips are organized into ranks and each rank is divided into multiple banks. Each memory bank is accessed one row (a chunk of data elements with consecutive memory addresses) at a time, and the entire row must be copied into a small buffer called the *row-buffer*. Figure 3 shows the high-level organization of a manycore system and its memory system (DRAM). Requests to different banks can proceed in parallel. As a result, their access latencies can be overlapped, improving DRAM throughput and leading eventually to high system performance. Bank-Level Parallelism (BLP) refers to the number of memory banks servicing multiple requests in parallel. In the context of tile scheduling, a critical question is whether the outstanding memory requests (last-level cache misses) originating from the concurrently-scheduled tiles can actually be serviced in parallel by different DRAM banks. If the requests in the DRAM controller buffers (which we call DRAM request buffers) are not to different banks, the amount of BLP achieved will be very low, thereby reducing the effectiveness of such techniques. In other words, while a carefully-selected tile size can improve cache performance, its impact on bank-level parallelism is not clear.

III. Problem Definition and Overview of the Proposed Scheme

To our knowledge, except for the work of Pai and Adve [32], which focuses on single-core machines, all prior tile scheduling strategies considered the parallelism only from the program perspective, and did not take the bank-level parallelism into account. Consequently, it is not clear how tiling affects BLP and whether a BLP-aware tile scheduling can bring additional performance improvements over a (default) scheduling that does not care about BLP. Consider Figure 2(c) again: if T_0 through T_3 happen to access a very small number of banks, significant memory stalls can be experienced. In contrast, if these tiles access different banks, we may have high levels of BLP. Motivated by this, the problem we address in this paper can be defined as follows:

> *How can a compiler schedule the tiles in a dependence-free set such that high levels of BLP can be achieved?*

Our proposed BLP-aware tile scheduling scheme can be used with any existing tiling strategy, and it consists of two steps. In this first step, which is called *Per-Tile Cache Miss Prediction*, we predict the iterations in each tile that will incur last-level cache misses, by employing the Cache Miss Equations [14]. Then, in the second step, which is called *Exploiting Bank-Level Parallelism*, we determine the set of memory banks that will be accessed in each tile (based on the loop indices of these iterations, the base address, and the cache/memory parameters), and propose two algorithms that handle the cases of small and large tile sizes, respectively, to exploit BLP. With a small tile size, the sequence of accessed memory banks in each tile will not affect the scheduling decision much since these bank accesses can be considered "concurrent". That is, as long as two tiles have the same memory bank to be accessed, scheduling these two tiles at the same logical time slot will hurt BLP since we have multiple accesses to the same memory bank around the same time. In contrast, with a large tile size, the duration (distance) between the first bank access and the last bank access in each tile could be long. As a result, the bank access order, in turn, affects the scheduling decision to maximize BLP. For example, even if two tiles have the same bank to be accessed, as long as the duration between the first access and second access to this bank is long enough (such that the first access is finished by the time when the second access starts), scheduling these two tiles at the same time slot can still achieve a high BLP. Figure 4 illustrates where our proposed compiler support fits in the relevant part of the compilation flow.

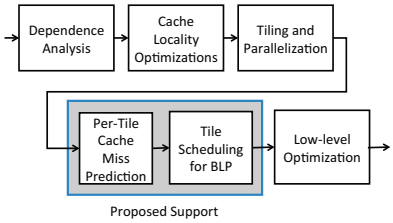

Fig. 4: The place of our proposed scheme in the compilation flow.

IV. Per-Tile Cache Miss Prediction

Cache Miss Equations (CME) [14] gives an accurate prediction of potential cache misses for a given loop nest that accesses array data. It consists of a set of affine constraints (denoted as $C(\vec{i})$) that an iteration $\vec{i}$ needs to satisfy if the accessed data elements (through array reference $\vec{r} = A\vec{i} + \vec{o}$) incur cache misses. These constraints collectively form a polyhedron. Therefore, determining whether a data access through $\vec{r}$ causes a miss or a hit is equivalent to determining whether, after substituting the corresponding $\vec{i}$ in the CME, the resulting polyhedron is non-empty.

Our per-tile cache miss prediction is based on CME. Specifically, given a tiled loop nest, CME can be used to identify all the data accesses that incur cache misses. In other words, by using CME, at each iteration $\vec{i}$, the set of accessed data elements that incur cache misses can be identified by $C(\vec{i})$. Furthermore, assuming that the lower and upper loop bounds for each tile T_k are $\vec{L}_k$ and $\vec{U}_k$, respectively, then by simply adding the following constraint to $C(\vec{i})$, the data elements in T_k that incur cache misses can also be identified:

$$\vec{L}_k \leq \vec{i} \leq \vec{U}_k. \tag{2}$$

In addition to this, for the purposes of this work, we enhanced CME to capture misses in a three-layer cache hierarchy; details of our CME enhancements are omitted due to lack of space.

V. Exploiting Bank-Level Parallelism

To exploit bank-level parallelism, the first problem is to determine the memory bank that will be targeted by a data access if it misses in the last-level cache (L3 in our case). Considering the memory address in terms of the array element size, the (virtual) memory address of a data element accessed through array reference $\vec{r}$ can be expressed as:

$$Mem(\vec{r}) = B + r_0 + \sum_{k=1}^{d-1} (m_{k-1} r_k), \tag{3}$$

where $\vec{r} = (r_0, r_1, \cdots, r_{d-1})^T$, B is the base address of the array, and m_k is the size of the array along the k-th dimension. In addition, most DRAM controllers employ a bank-level interleaving (round-robin) strategy [4]. Under this scheme, consecutive memory blocks in physical address space, typically of the size of a row-buffer (or page), are allocated to different banks. As a result, the memory bank to which a data element accessed through array reference $\vec{r}$ is mapped can be expressed as:

$$Bank(\vec{r}) = \lfloor Mem'(\vec{r})/R \rfloor \, mod M, \tag{4}$$

where R is the row-buffer size, M is the total number of memory banks, and $Mem'(\vec{r})$ is the physical memory address of that data element (accessed through $\vec{r}$).

To determine $Bank(\vec{r})$, we need to know the mapping between the physical and virtual pages (addresses). In most page allocation schemes, the OS maintains a free list, which holds virtual pages that are not mapped into any physical address space. Whenever a running process needs a new memory space, the OS allocates free pages from this list and updates the list accordingly. In Solaris for example, new pages are allocated by calling the page_create_va() function, which accepts the virtual address of the location to which the page is going to be mapped as an argument; then, the virtual-to-physical coloring algorithm can decide which color to take physical pages from. Therefore, in this work, we *enforce* a virtual-to-physical page mapping (by changing the page_create_va() function) such that, when assigning a physical address space to a page with the given virtual address, we require that the assigned physical address and the calculated virtual address are mapped to the same memory bank. We only enforce this mapping for accesses that incur cache misses, and only if doing so does not create any additional page faults. In our experiments, we did not observe an increase in conflict misses.

With this virtual-to-physical page mapping, we have:

$$Bank(\vec{r}) = \lfloor Mem(\vec{r})/R \rfloor \, mod M = \lfloor Mem'(\vec{r})/R \rfloor \, mod M, \tag{5}$$

At this point, with Expressions 1- 5 and CME, the memory bank accesses within each tile can be determined. To exploit the bank-level parallelism, our next problem is to schedule the tiles based on these bank accesses.

A. Bank Access-Order Oblivious Scheduling

If the tile size is small enough, all the memory bank accesses within each tile will take place in a short period of time. Therefore, if we schedule two tiles at the same time slot but on different cores, as long as there exist accesses to the same bank in both tiles, *conflicting bank accesses* (i.e., multiple accesses to the same bank around the same time) will occur. The tile scheduling we propose to handle this case is called *Bank Access-Order Oblivious Scheduling* since we do not care about the bank access order within a tile. Let us assume that in each tile T_k, a set of banks, B_k, will be accessed, and there are N cores in the target architecture. To maximize BLP, the key observation is that, at each time (scheduling) slot, we should cover *as many different banks as possible*. As a result, our BLP optimization problem can be abstracted as follows:

Input: *A collection of sets* $B = B_1, B_2, \cdots, B_n$ *and* N.

Goal: *Find a subset* $B' \subseteq B$, *such that* $|B'| = N$ *and the number of covered elements* $|\bigcup_{B'_i \in B'} B'_i|$ *is maximized.*

Clearly, this is a maximum coverage problem [12], which is known to be NP-hard. It has been shown that a greedy algorithm is essentially the best-possible polynomial time

Algorithm 1 Bank Access-Order Oblivious Scheduling.

```
1: Tiles ← All the tiles in a dependence-free set;
2: Selected ← ∅;
3: /* assuming that we have N cores*/
4: for each logical time slot t_0 do
5:     calculate σ_cvd(T_i) for each tile T_i in Tiles;
6:     select T_k with the largest σ_cvd value as the first tile to be scheduled at
       t_0;
7:     Selected ← Selected ∪ T_k;
8:     Tiles ← Tiles − T_k;
9:     NumSelected ← 1;
10:    while NumSelected ≠ N do
11:        CVD ← σ_cvd(Selected, T_i);
12:        calculate CVD for each tile T_i in Tiles;
13:        select T_p with the largest value of CVD as the tile to be scheduled
           at t_0;
14:        Selected ← Selected ∪ T_p;
15:        Tiles ← Tiles − T_p;
16:        NumSelected ← NumSelected + 1;
17:    end while
18: end for
```

approximation algorithm for the maximum coverage problem [12]. We also explore a greedy (tile) scheduling algorithm in this work. Specifically, at each stage, we choose a tile which contains the largest number of uncovered bank accesses. To achieve this, we employ a concept called *bank vector*. This vector, defined separately for each tile, has N entries, and each entry (which corresponds to a bank) can be either 1 or 0. A "1" in the k-th entry indicates that the k-th bank will be accessed by this tile, and a "0" indicates that the k-th bank will not be accessed in this tile. A bank vector can be used to calculate the number of covered banks for any number of tiles in a very efficient way. Given two bank vectors $\vec{b}_1 = (b_{1,0}, b_{1,1}, \cdots, b_{1,N-1})^T$ and $\vec{b}_2 = (b_{2,0}, b_{2,1}, \cdots, b_{2,N-1})^T$ (corresponding to tiles T_1 and T_2, respectively), the number of covered banks ($\sigma_{cvd}(T_1, T_2)$) can be expressed as:

$$\sigma_{cvd}(T_1, T_2) = \sum_{i=0}^{N-1} (b_{1,i} | b_{2,i}), \tag{6}$$

where "|" denotes the bitwise-OR operator. Without loss of generality, let us now assume that, at a given stage, p tiles have already been scheduled, and we need to choose a tile that accesses the largest number of uncovered banks. Based on Expression 6. one can make this selection by first calculating $\sigma_{cvd}(T_k, T_1, T_2, \cdots, T_p)$ for each of the remaining tiles, and then choosing the one that has the largest value of $\sigma_{cvd}(T_k, T_1, T_2, \cdots, T_p)$. The pseudo-code of our bank access-order oblivious scheduling is given in Algorithm 1. Assuming that we have P time slots for tile scheduling, and there are Q tiles in the dependence-free set, then, at each time slot t_0, we need to perform $Q + (Q-1) + \cdots + (Q-N)$ comparisons in order to select the largest σ_{cvd}. Since usually $Q >> N$, the overall complexity of Algorithm 1 can be given as $O(PNQ)$.

B. Bank Access-Order Conscious Scheduling

The tile size could be large as well, depending on the tiling strategy used by the compiler as well as the cache parameters. In this case, we cannot assume the bank accesses within a tile will take place in a short time period (concurrent). Consequently, we need to take the bank access order into account maximize the BLP. As an example, let us assume that, in Figure 5, two tiles (T_1 and T_2) have accesses to the same set of banks, *Bank*1 and *Bank*2, but in different orders. In the small tile-size case, these two tiles will not be scheduled at the same time slot since they access the same set of banks at around the same time (t_0), as illustrated in Figure 5(b). However, in the large tile-size case, as shown in Figure 5(a), if *Bank*1 and *Bank*2 (in T_1) are accessed at times t'_0 and t'_1, respectively, where $t'_0 << t'_1$, and in T_2, they are accessed at t'_1 and t'_0, then these two tiles indeed can be scheduled at the same slot since t'_0 and t'_1 are far away from each other. By the time the bank accesses at t'_1 start, the bank accesses at t'_0 would have been finished. As a result, high levels of BLP can be achieved.

Algorithm 2 Calculating $n(\tau_i, \tau'_i)$.

```
1: n(τ_i, τ'_i) ← 0;
2: t ← t_min;
3: t' ← t'_min;
4: while t ≠ t_max ∧ t' ≠ t'_max do
5:     if t ≤ t' then
6:         if t' − t < w then
7:             n(τ_i, τ'_i)++;
8:         end if
9:         t ← t_next;
10:    else
11:        if t − t' < w then
12:            n(τ_i, τ'_i)++;
13:        end if
14:        t' ← t'_next;
15:    end if
16: end while
```

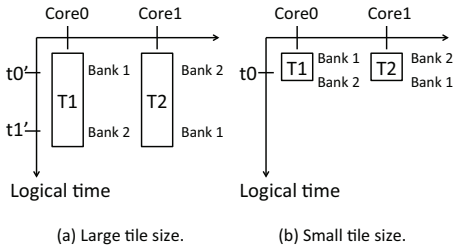

Fig. 5: (a) With a large tile size, the given scheduling does not have bank conflicts. (b) With a small tile size, the given scheduling has bank conflicts.

Our proposed support employs a "time window" (measured in terms of "loop iterations") to handle the large tile-size case: *only* the accesses to the same bank from different tiles within a time window w will be considered *conflicting bank accesses*. To maximize BLP, we propose to choose, at each stage, the tile that has the *minimum* number of conflicting bank accesses with the tiles that have been selected.

Without loss of generality, let us assume that, at a given point in scheduling, tiles $T_1, T_2, \cdots, T_p$ have already been scheduled at a time slot, and we need to find another tile T_k from among the remaining tiles that can also be scheduled

at the same time slot. To make this selection, we can first calculate the total number of conflicting bank accesses among $T_1, T_2, \cdots, T_p$ and each remaining tile (based on the time window), and then choose the one with the minimum conflicts. Specifically, we attach a "time bucket" (denoted as τ_i) to each entry b_i in the bank vector. Each time bucket consists of a set of "time stamps", and each time stamp is the position of the iteration in the tile that incurs the bank access. For example, if the k-th iteration in the given tile incurs a bank access, then the corresponding time stamp will be k. Note that, since multiple iterations may incur the same bank access, there may exist multiple time stamps for b_i.[2] With this information, we have the following observation:

Given $T_1 : \langle b_0, \tau_0 \rangle$, $\langle b_1, \tau_1 \rangle$, $\cdots$, $\langle b_{N-1}, \tau_{N-1} \rangle$, and $T_2 : \langle b'_0, \tau'_0 \rangle$, $\langle b'_1, \tau'_1 \rangle$, $\cdots$, $\langle b'_{N-1}, \tau'_{N-1} \rangle$, for every pair of b_i and b'_i that satisfies $b_i \,\&\, b'_i = 1$, where & is the bitwise-AND operator and $0 \leq i \leq N-1$, if there exist $t_i \in \tau_i$ and $t'_i \in \tau'_i$ such that $|t_i - t'_i| < w$, then in T_1 and T_2, the accesses to Bank i cause a conflicting bank access.

For example, let us assume that $T_1 : \langle 1,0 \rangle$, $\langle 0,0 \rangle$, $\langle 1,2 \rangle$, $\langle 0,0 \rangle$, and $T_2 : \langle 1,2 \rangle$, $\langle 0,0 \rangle$, $\langle 1,7 \rangle$, $\langle 0,0 \rangle$. This indicates that in T_1, *Bank*0 and *Bank*2 will be accessed at time stamps 0 and 2, respectively, and in T_2, *Bank*0 and *Bank*2 will be accessed at time stamps 2 and 7, respectively. Therefore, we have $\langle b_0, \tau_0 \rangle = \langle 1,0 \rangle$, $\langle b_2, \tau_2 \rangle = \langle 1,2 \rangle$, and so on. If the window size is 3 (iterations), then the two accesses to *Bank*0 will fall into this windows since $|2-0| < 3$, and the two accesses to *Bank*2 will not fall into this window since $|7-2| > 3$. As a result, there will be one conflicting bank access between T_1 and T_2. Based on this discussion, the total number of conflicting bank accesses between T_1 and T_2 (denoted as $\sigma_{cft}(T_1, T_2)$) can be expressed as:

$$\sigma_{cft}(T_1, T_2) = \sum_{i=0}^{N-1} (b_i \& b'_i) \times n(\tau_i, \tau'_i), \tag{7}$$

where $n(\tau_i, \tau'_i)$ denotes the number of pairs of t_i and t'_i that satisfy $t_i \in \tau_i$, $t'_i \in \tau'_i$, and $|t_i - t'_i| < w$. The key to calculating $\sigma_{cft}(T_1, T_2)$ is to determine $n(\tau_i, \tau'_i)$. Let t_{max} and t_{min} denote the largest and smallest time stamps, respectively, in each time bucket.[3] Then, $n(\tau_i, \tau'_i)$ can be calculated by traversing the time stamp lists in τ_i and τ'_i once, as shown in Algorithm 2. Assuming that the total number of time stamps in τ_i and τ'_i is N_0, then the complexity of Algorithm 2 is $O(N_0)$. This algorithm can be extended to handle the case of calculating $\sigma_{cft}(T_1, T_2, \cdots, T_k)$. Details are omitted due to lack of space. The pseudo-code of our bank access-order conscious scheduling is given in Algorithm 3. Similar to Algorithm 1, assuming that $Q >> N, Q >> N_0$, the complexity of Algorithm 3 can be given as $O(PNQ)$, where P is the number of time (scheduling) slots, Q is the number of tiles in the dependence-free set, and N is the number of cores.

[2]This is primary the reason why we have the time bucket.

[3]The time stamps in each time bucket are already ordered since these time stamps are determined by traversing the iterations in each tile.

Algorithm 3 Bank Access-Order Conscious Scheduling.

```
1: Tiles ← All the tiles in a dependence-free set;
2: Selected ← ∅;
3: /* assuming that we have N cores*/
4: for each logical time slot t_0 do
5:    randomly select T_k as the first tile to be scheduled at t_0;
6:    Selected ← Selected ∪ T_k;
7:    Tiles ← Tiles − T_k;
8:    NumSelected ← 1;
9:    while NumSelected ≠ N do
10:      CFT ← σ_cft(Selected, T_i);
11:      calculate CFT for each tile T_i in Tiles with T_k;
12:      select T_p with the smallest value of CFT as the tile to be scheduled
         at t_0;
13:      Selected ← Selected ∪ T_p;
14:      Tiles ← Tiles − T_p;
15:      NumSelected ← NumSelected + 1;
16:   end while
17: end for
```

VI. Extension to Memory Controller-Level Parallelism (CLP)

So far, in our discussion, we focused exclusively on maximizing bank-level parallelism. However, there are at least two other metrics that one might want to consider when maximizing memory system performance: *memory controller-level parallelism* (CLP) and *row-buffer locality* (RBL). CLP refers to the number of memory controllers that serve memory requests concurrently. It is different from BLP as neither of them necessarily implies the other. For example, an application may have a relatively high BLP, but if all the banks used are managed by the same memory controller, CLP will be only 1. Similarly, we can achieve a maximum possible CLP (4 in our default system), but if only one bank is accessed from each MC, the resulting BLP with be only 4.

On our NoC-based manycore system, an off-chip data access (main memory access) needs to travel through the on-chip network, spending considerable time within the chip (in addition to the memory access latency). In addition, it contends with on-chip (cache) accesses as both use the same NoC resources. Therefore, if the accessed banks in different tiles reside in the same memory controller at a given time slot, then the network congestion near that memory controller could be much higher, compared to other parts of the network. This will cause unbalanced network traffic and therefore increase both the on-chip and off-chip access latencies.

To avoid such a situation, our two proposed tile scheduling algorithms can be extended to support CLP. Observe that, in both the algorithms, we select N tiles to be scheduled at each time slot, based on the value of either CVD or CFT for each candidate tile. Therefore, to support CLP, our idea is to add additional constraints on the tile selection phase such that the selected tiles have accesses to the banks that reside in different memory controllers as much as possible. In our current compiler implementation, we associate a counter with each memory controller at each time slot, which counts the number of banks that reside in that memory controller at that time slot. The mapping between the banks and memory controllers is given as input and is exposed to the compiler.

When making the tile selection decision the tiles, if multiple tiles have the same value of CVD (Algorithm 1) or CFT (Algorithm 3), we always select the ones that have the accesses to the banks in the memory controller with the smallest counter.

In other words, by exposing the bank-to-memory controller mapping to the compiler, for each L3 miss, we figure out the memory controller (MC) that will be accessed. From an CLP perspective, we would want to select tiles (to schedule at the same time slot) such that the misses are distributed across available MCs as evenly as possible. Clearly, this goal can conflict with BLP maximization. To handle this problem, our approach operates as follows. If it is possible to select a tile (to schedule) that maximizes both BLP and CLP at the same time, we select that tile. If not, our current implementation favors CLP over BLP (note that favoring BLP over CLP would generate very similar results to a pure BLP optimization). As will be presented later in Section VIII, we found that, in most applications, this CLP-aware version is able to optimize both BLP and CLP, and it brings significant additional benefits (mostly due to reduction in on-chip network congestion) over pure BLP optimizations in three applications.

VII. Extension to Row-Buffer Locality (RBL)

In addition to BLP and CLP, one would also want to exploit row-buffer locality (RBL) for tile scheduling. Recall that each memory bank is accessed one row (a chunk of data elements with consecutive memory addresses) at a time. If an *open page policy* is used, the same row is kept in the row-buffer after the initial request, allowing multiple accesses to the same row. The data in the row-buffer only changes if a different row is requested. Memory accesses that find their row already in the row-buffer are called "row-buffer hits" and can be significantly faster than "row-buffer misses", which must wait for the row to be copied into the buffer. *Row-buffer locality* refers to the reuse of a memory row while its contents are in the row-buffer. If there are multiple requests to the same bank, a better row-buffer locality (i.e., higher row-buffer hit rate) can be achieved if the requested data elements can be found in the same memory row.

RBL can be exploited in two ways within our compiler framework. First, at a given time slot t, assume that we have selected k tiles and need to select the $k+1$-th tile that will also be scheduled at t from multiple candidates with the same CVD (Algorithm 1) or CFT (Algorithm 3). If these candidates have the same number of conflicting bank accesses with the selected k tiles, then we prefer to choose the tiles that maximize row-buffer locality (row-buffer hits). However, exploiting row-buffer locality in this way implies a very strict constraint: between the two conflicting bank accesses (where the accessed data reside in the same row-buffer), there must be no other access to the same bank. Since in multithreaded applications, the exact iteration execution order or data access order among the tiles (at the same scheduling time slot) is hard to predict, it also becomes hard to determine if this restrictive condition can be satisfied or not for a given set of bank accesses. Therefore, we did not opt for this implementation in this work.

For the second implementation option, let us assume that the tiles that need to be scheduled at time slot t have already been selected. Now, we move to the selection of tiles at time slot $t+1$. Let RB denote the set of row-buffers in different banks that have been most recently accessed at time slot t. From the RBL perspective, we prefer to select the tiles whose first off-chip (bank) access is due to a request to data elements residing in the row-buffers in RB. Specifically, let us assume that k tiles have been selected at $t+1$ and we need to select the $k+1$-th tile that will also be scheduled at $t+1$. If we have multiple candidates with the same CVD (Algorithm 1) or CFT (Algorithm 3), then we can always choose the tile that is next to any of the selected tiles at time slot t in the iteration space. The reason is that, in most affine loops, it is more likely that successive iterations access data elements in consecutive memory locations. Therefore, if the tiles are chosen this way, the selected tiles are more likely to access the same row-buffers as the tiles that have been selected at t.

Similar to the CLP case, if we cannot achieve both BLP/CLP and RBL, we need to favor one of them over the other. Our current implementation favors RBL over BLP if we cannot achieve both. In global order, we consider RBL first, then CLP, then BLP.

VIII. Experimental Evaluation

The compiler support discussed in this paper is implemented in the Open64 infrastructure [3]. We use GCC version 4.8.3 and OpenMP version 3.0 without dynamic load balancing. We used the GEM5 [1] tool-set to perform our experiments, and collected detailed statistics regarding BLP, CLP, and RBL (which is not possible in real hardware). The default 4×8 manycore configuration simulated in this work is given in Table I. Later we will present results where the values of some of the parameters shown in Table I are modified. We used the benchmarks from the SPECOMP suite [8] with the ref input set. For each benchmark, we tested the following four versions:

- *Base.* This is the base version we have. It does not include any optimizations specific to BLP, CLP or RBL, but does apply tiling.
- *BLP-opt.* This version performs the BLP-aware scheduling explained in Section V.
- *BLP-CLP-opt.* This version optimizes for both BLP and CLP, as explained in Section VI.
- *BLP-CLP-RBL-opt.* As explained in Section VII, this version combines BLP, CLP and RBL optimizations.

These versions only differ from one another in terms of what they do for BLP, CLP and/or RBL; they employ the same code parallelization and tiling strategy, where tile shapes and sizes are selected to minimize last-level cache misses as well as inter-tile dependences. Table II gives our benchmarks and their important characteristics. All the numbers in this table are for the *base version* explained above. Each benchmark tested (for any version above) has been compiled using the

TABLE I: Architecture specification.

Core	32 cores, 2GHz, out-of-order, 15 stages, decode/retire up to 3 instr, issue/execute up to 8 microinstructions, 256-entry reorder buffer, 32-entry load/store buffer, 256 physical registers, 4-entry BTB.
NoC	4 × 8, 4 VCs, 4 flits/VC, 2 cycle router latency
Cache Hierarchy	3 layers, L1 = private; 32KB per core; 2-cycle latency; 2-way; 64-byte line size. L2 = private; 128KB per core; 8-cycle latency; 4-way; 128-byte line size. L3 = shared (static NUCA); 512KB per core; 12-cycle latency, 32-way; 128-byte line size.
Memory System	4 on-chip memory controllers (MCs) placed in 4 corners of the chip, 256-entry L3 MSHR, 2 ranks per channel, 8 banks per rank, all DDR3 parameters from Micron Datasheet [2], FR-FCFS scheduling policy, page-interleaving mapping policy, row-buffer size = 4KB.
Prefetcher	stream prefetcher (in L3 controller), with 32 stream entries; prefetch distance = 64; prefetch degree = 4; prefetch buffer size = 64 entries.

TABLE II: Benchmark codes used in our evaluation.

Name	L3 Misses (Actual)	L3 Misses (Predicted)	BLP	CLP	RBL(%) (Row-Buffer Hit Rate)	IPC
wupwise	14.1	13.9	6.7	2.8	27.6	8.9
swim	20.9	20.2	9.6	3.2	21.3	12.2
mgrid	21.2	19.8	7.2	2.2	18.8	6.7
applu	13.5	13.1	9.2	3.0	34.4	8.2
galgel	22.6	21.0	11.3	2.9	28.7	10.2
equake	18.6	14.6	6.2	1.8	19.6	9
apsi	11.5	8.8	8.6	3.2	28.2	12.4
gafort	16.1	15.7	7.9	2.7	32.2	7.2
fma3d	27.4	26.5	13.3	3.5	43.3	11
art	18.3	17.6	8.8	3.2	24.4	6.2
ammp	13.6	13.1	5.9	2.6	33.7	8.1

O3 optimization level with all data locality optimizations and low-level code optimizations turned on. For the simulations, we fast-forwarded 500 million instructions and simulated the next 1.5 billion instructions. These experiments varied between 44.3 seconds and 2.6 minutes. Based on some preliminary experiments we performed (details are omitted for space concerns), we used an iteration count of 100 as the threshold between small tiles and large tiles. That is, if a tile contains 100 or more (loop) iterations, we tag it "large"; otherwise, we tag it "small", and use the appropriate BLP optimization algorithm in each case. We also made experiments with (cache) block granularity interleaving (instead of page granularity); the IPC savings we observed were about 2% higher than the page-interleaving case.

A. Results with the Default Values of Simulation Parameters

a) BLP Results. : We start by presenting the BLP results with these different versions. On average, with respect to the base version, BLP-opt, BLP-CLP-opt, and BLP-CLP-RBL-opt bring 17.1%, 15.8%, and 9.5% BLP improvements, respectively. The results plotted in Figure 6(a) indicate that the BLP-opt version improves over the base version in seven of our eleven benchmarks programs. In four benchmarks on the other hand, our approach could not bring much benefit. In *galgel* and *art*, the last-level cache (L3) misses are clustered into certain address regions, which prevents our scheduler from achieving its goal fully. Because of this clustering, even working with a large number of banks does not help with these codes. In *apsi* and *equake* on the other hand, we noticed that our compiler was not very successful in estimating L3 misses (see Table II), mostly due to irregular data accesses. As a result, our scheduler does not do a very good job with these two codes either. We further observe from Figure 6 that BLP-CLP-opt generates very similar results to BLP-opt;

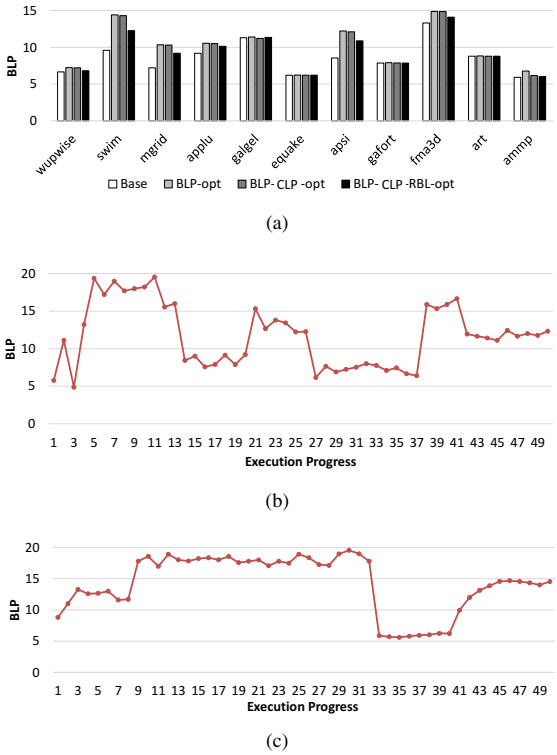

Fig. 6: (a) BLP values with different schemes.

the difference between them is less than 2% in all but one benchmark (*ammp*). This result is not surprising, because BLP-CLP-opt tries to optimize CLP while maintaining the best BLP, and this results in only minor deviations from BLP-opt. However, the difference between BLP-CLP-RBL-opt and BLP-opt is more pronounced, primarily because the former tries to trade off bank-level parallelism for row-buffer locality (row-buffer hit rate).

Figure 6(b) shows the BLP variation in the case of *apsi* over the execution period for BLP-opt. Comparing this plot with the one given in Figure 1(c) reveals that our scheduling strategy improves BLP of this application in each major phase (loop nest). Figure 6(c) gives the same plot for *swim*. Comparing this graph with Figure 1(d), we see that BLP has been improved in all phases except one (between epochs 33 and 40).

b) CLP Results. : Next, we discuss how our approach affects CLP of our benchmarks. Recall from our discussion in Section VI that a high CLP indicates a good distribution

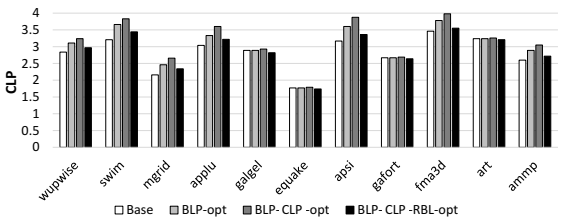

Fig. 7: CLP values with different schemes.

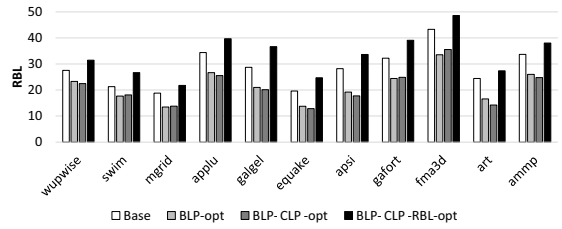

Fig. 8: RBL values with different schemes.

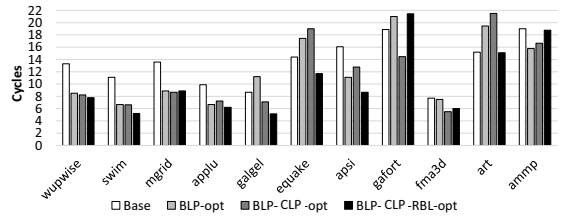

Fig. 9: Cycles-per-load (CPL) values with different schemes.

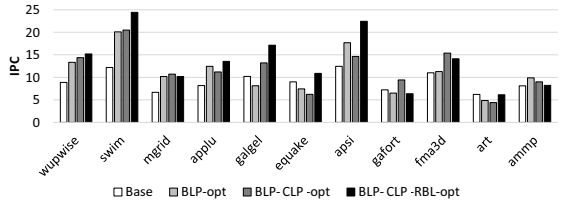

Fig. 10: IPC values with different schemes.

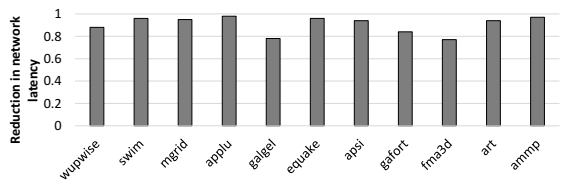

Fig. 11: Network latency with BLP-CLP-opt, normalized with respect to the base version.

of memory accesses over the NoC as well as across MCs. Furthermore, CLP and BLP do not necessarily correlate well in all cases. It can be seen from Figure 7 that BLP-opt, BLP-CLP-opt, and BLP-CLP-MLP-opt bring an average CLP improvement of 7.6%, 12.4%, and 3.5%, respectively, over the base version. In the case of BLP-opt, balancing accesses across banks helps (to a certain extent) to balance them across memory controllers as well. That is, even though BLP-opt's goal is to optimize BLP, maximizing bank-level parallelism tends to distribute concurrent accesses across different MCs. However, one can also see that the memory controller-level parallelism is maximized with the BLP-CLP-opt version (12.4% improvement over the base version). This result means that it may be possible for many multithreaded applications to optimize BLP and CLP together. Finally, while BLP-CLP-RBL-opt improves over the base version as far as memory-level parallelism is concerned, the percentage savings it brings are not as good as BLP-opt or CLP-BLP-opt. This is because it is more oriented towards improving row-buffer hit rates while maintaining good BLP, and as a result, CLP can suffer.

c) RBL Results. : The results in Figure 8 show that BLP-CLP-RBL-opt improves over the base version in all applications, generating an average of 17.8% improvement in row-buffer hit rates. In contrast, the other two versions, BLP-opt and BLP-CLP-opt, result in lower hit rates. This is due to the fact that the primary goal of these two versions is to improve memory access concurrency and this in most cases results in a degradation in row-buffer hit rates.

d) Performance Improvement. : Since BLP-opt, BLP-CLP-opt, and BLP-CLP-RBL-opt have different impacts on bank-level parallelism, memory-level parallelism and row-buffer hit rates, their overall impact on memory performance with respect to each other is not trivial to predict. The overall impact of these different versions on memory access latency (cycles-per-load) and IPC are plotted in Figures 9 and 10, respectively. One can make several key observations from these plots. First, the relative contributions of BLP and RBL to the overall performance are different for different benchmarks; therefore, a high BLP may not completely translate to a high performance if it is not also accompanied by a high RBL. Second, we see that the BLP-opt version improves performance (over the base version) in seven of our 11 applications, and generates worse results than the base case in *galgel*, *equake*, *gafort* and *art*, due to, primarily, the deterioration it causes in row-buffer locality. Third, BLP-CLP-opt performs similar to BLP-opt, but it generates better results than the latter in *galgel* and *gafort*, two of the benchmarks where the base version performed better than BLP-opt. In fact, it can be observed that BLP-CLP-opt performs quite well in *galgel*, *gafort* and *fma3d*. To explain why this happens, we present in Figure 11 the reduction in network latency (cycles spent in the NoC over the entire execution numbers for different versions) when using BLP-CLP-opt. We note in this graph the significant improvement in these three applications. In other words, distributing accesses across the MCs helps to reduce the network congestion in these three applications. Finally, BLP-CLP-RBL-opt generates better results than the base case in

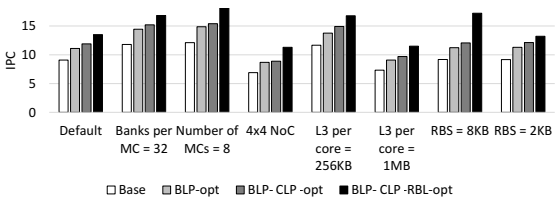

Fig. 12: Results from the sensitivity experiments.

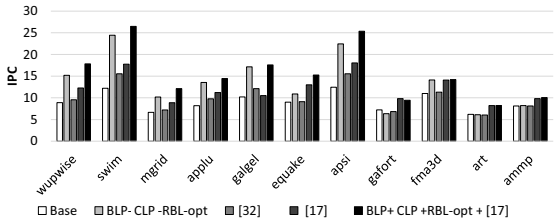

Fig. 13: The IPC results with [32], [17], BLP-CLP-RBL-opt version and the base scheme.

all cases except *art* and *ammp*; in those two benchmarks they perform very similarly. That is, balancing BLP and RBL pays off in most of the applications we tested. In fact, BLP-CLP-RBL-opt significantly outperforms BLP-opt and BLP-CLP-opt in *apsi*, *equake* and *swim*, in terms of both memory access latency and IPC.

B. Results from the Sensitivity Experiments

In this section, we vary the default values of some of the parameters in Table I, and conduct a sensitivity analysis of our proposed approach. Due to space concerns, we present average results across benchmarks, instead of results for each benchmark individually, and report only IPC results. In Figure 12, the first group of bars gives the average IPC values with the default values of our simulation parameters (Table I). The first parameter we study is the number of banks per MC. Recall that the default value used in our experiments so far was 16. When we increase the number of banks to 32 (per MC), all the tested versions take advantage of it and IPC values increase. The second parameter we study is the number of MCs, and we see that doubling their number (while keeping the number of banks per MC at 8) generates similar results to doubling the number of banks. We next evaluate a smaller system with 16 nodes (4×4), and we observe, as expected, a reduction in our savings. The experiments with different L3 capacities indicate that the effectiveness of our approach increases with reduced L3 capacity. This is because a reduced L3 capacity increases the chances for better memory-level parallelism. Finally, reducing the row-buffer size helps, as can be expected, the BLP-CLP-RBL-opt version most, because the original performance with smaller buffer size is very bad, and consequently the RBL optimization (in BLP-MLP-RBL-opt) has more scope for improvement.

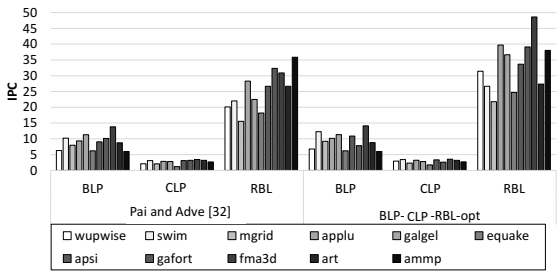

Fig. 14: BLP, CLP and RBL metrics for the scheme in [32].

C. Comparison against Prior Schemes

In this section, we compare our schemes against two previously-published works. The first scheme [32] is a compiler-based scheme targeting single-core systems that *clusters* cache misses, which does not take into account banks or memory controllers, and implicitly assumes that clustered misses will result in high levels of memory access concurrency. The second scheme [17] is a hardware-based strategy, which tries to *balance row-buffer locality and memory parallelism*. On the RBL side, the approach proposed by Jeong et al. [17] partitions the memory banks between cores in an attempt to isolate their access streams and minimize interferences. Since doing so may hurt BLP, it also employs memory sub-ranking to effectively increase the number of banks.

The IPC results with these two schemes as well as our BLP-CLP-RBL-opt version and the base scheme are plotted in Figure 13. Since the compiler scheme in [32] is designed for single-core systems, we applied it to computations mapped to each core independently, by simply changing the traversal order of the iterations in each tile. Our first observation is that BLP-CLP-RBL-opt performs better than this scheme in all applications except *gafort*. This is because clustering cache misses does *not* necessarily improve bank-level parallelism. While a high number of misses is necessary for high memory parallelism, that may not be sufficient. In other words, simply increasing the number of concurrent memory accesses does not necessarily lead to high BLP (when they are not spread over a large number of banks). This is because how these misses are mapped to available MCs and banks can also play an equally important role. In addition, the compiler approach in [32] does not consider row-buffer locality explicitly (except that row-buffer locality can be improved as a side-effect of cache locality optimizations such as tiling). To explain this better, we present in Figure 14 the BLP, CLP and RBL metrics for the scheme in [32] (the values of the same metrics under BLP-CLP-RBL-opt are also reproduced here for ease of comparison). Note that, in all these metrics, [32] performs worse than BLP-CLP-RBL-opt, which explains the IPC difference between them. We next observe that BLP-CLP-RBL-opt outperforms the hardware scheme in [17] in six of the applications, with an average IPC improvement of 30.9%. In four applications on the other hand, [17] generates

29.3% better results (on average) than BLP-CLP-RBL-opt, and in one application (*fma3d*), they generate very similar results. Basically, in applications where compiler has limited scope for optimization (e.g., *gafort*), a pure hardware approach is certainly a better option. However, in applications where compiler can successfully schedule tiles (e.g., *swim*), BLP-CLP-RBL-opt brings significant benefits over the hardware approach. It is also interesting to see, from the last bar for each application, that a combination of BLP-CLP-RBL-opt and [17] maximizes the savings in most of our benchmarks.

IX. Related Work

Iteration Space Tiling. Early compiler-based studies looked at loop tiling (also known as loop blocking) to exploit multiple levels of cache and improve parallelism [44], [16], [22], [36], [43], [25]. However, none of these prior compiler works focused on bank-level parallelism. More recently, the work of Baskaran et al. [10] generates a task dependence graph at runtime that is used to schedule tiles for parallelism. Liu et al. [26] focus on cache locality in the context of a multi-level cache hierarchy, but do not consider memory-level parallelism either. Zhou et al. [49] improve on overlapped tiling, which combines loop tiling and loop fusion to eliminate synchronization overhead, but introduces a lot of redundant computation. Pai and Adve [32], the scheme considered in Section VIII-C, use tiling to hide memory latency by clustering misses and to exploit locality. In contrast to our work, they do not take into account the target memory architecture, and do not target BLP exclusively.

Memory-Level Parallelism. Many previous works considered hardware designs to improve memory-level parallelism, which are not necessarily incompatible with our compiler optimization since they target runtime performance. Chen et al. [11] showed that for multicores, CLP was more important than row-buffer hit rate for performance. Rixner [37] suggested using virtual channels in memory for CLP. Qureshi et al. [35] proposed a cache replacement policy that considers the effect on CLP in deciding which lines to evict. Phadke and Narayanasamy [34] combine different types of memory optimized for latency, bandwidth, or power.

Bank-Level parallelism. Jeong et al. [17] use bank partitioning to improve row-buffer hit rate while maintaining bank-level parallelism. Kim et al. [21] take advantage of the fact that DRAM banks are already split into multiple subarrays with local row-buffers. Other similar papers looked at memory rank partitioning [5], [6], [48], [41], bank partitioning [29], [27], [45], and memory channel partitioning [30], [37]. Park et al. [33] use randomness to allocate page frames in order to improve BLP, because regular access patterns from multiple threads can worsen row-buffer conflicts. A similar paper by Zhang et al. [47] described causes of row-buffer conflicts when using a page interleaving policy. Their work improved row-buffer hit rates using permutations to preserve data locality while distributing pages to reduce conflicts.

Row-buffer Locality and Memory Controller-Level Parallelism. Many previous papers have focused on memory request scheduling to improve bank-level parallelism and row-buffer hit rates [20], [31], [15], [9], [50], [24]. Memory scheduling works on memory requests in the order they arrive from the last-level caches, whereas compiler schemes such as ours attempt to reorder memory accesses in the application itself. Lee et al. [24] try to reduce interference between reads and writes in the memory channel by prioritizing writebacks that will hit in the row-buffer. Mutlu and Moscibroda [31] propose a batch scheduler that uses both row-buffer locality and BLP to make scheduling decisions. Sudan et al. [39] try to improve row-buffer locality by decreasing the OS page size and moving frequently accessed pages into the same row.

Compiler Support for Multicores/Manycores. Kandemir et al. [18] develop a compiler optimization that takes the cache hierarchy into account so that threads that share data are placed onto cores that share caches. Lu et al. [28] describe a compile-time data layout scheme to improve locality in non-uniform cache architectures. Sung et al. [40] use an array tiling data layout to improve CLP of grid-based applications. These works are largely orthogonal to our paper.

Real-Time Computing. The domain of real-time computing has also studied memory performance as it contributes to latency. Yao et al. [46] improved memory utilization in a real-time multicore system using TDMA scheduling by allowing early memory accesses when they do not interfere with computation. Anderson et al. [7] developed a cache-aware scheduling strategy for real-time applications on multicores.

X. Concluding Remarks

In this paper, we present a compiler-based tile scheduling strategy to maximize BLP of multithreaded applications running on manycores. We discuss and evaluate two complementary compiler algorithms, one for small tile sizes and the other for large tile sizes. Our experiments indicate that the proposed scheduling strategy can result in average BLP improvements of 17.1%, with respect to the base execution. We also extend our compiler strategy to consider both memory controller-level parallelism and row-buffer locality (in addition to BLP), bringing even higher improvements. Finally, we compare our approach against two prior studies (one compiler- and one hardware-based) and discuss pros and cons of each scheme.

XI. Acknowledgments

This work is supported in part by NSF grants 1439057, 1439021, 0963839, 1409095, 1213052, and 1205618, as well as grants from Microsoft and Intel.

References

[1] "Gem5." [Online]. Available: http://gem5.org
[2] "Micron Datasheet." [Online]. Available: http://www.micron.com/
[3] "Open64." [Online]. Available: http://www.open64.net/
[4] "Intel 64 and ia-32 architectures optimization reference manual," 2012. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
[5] J. H. Ahn *et al.*, "Future scaling of processor-memory interfaces," in *Proc. of SC*, 2009.
[6] J. H. Ahn *et al.*, "Improving system energy efficiency with memory rank subsetting," *ACM Trans. Archit. Code Optim.*, 2012.

[7] J. Anderson *et al.*, "Real-time scheduling on multicore platforms," in *Proc. of RTAS*, 2006.

[8] V. Aslot *et al.*, "SPEComp: A new benchmark suite for measuring parallel computer performance," *OpenMP Shared Memory Parallel Programming*, 2001.

[9] R. Ausavarungnirun *et al.*, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," *SIGARCH Comput. Archit. News*, 2012.

[10] M. M. Baskaran *et al.*, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors," in *Proc. of PPoPP*, ser. PPoPP '09, 2009.

[11] L. Chen *et al.*, "A study of leveraging memory level parallelism for dram system on multi-core/many-core architecture," in *Proc. of TrustCom*, 2013.

[12] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[13] W. Ding *et al.*, "Reshaping cache misses to improve row-buffer locality in multicore systems," in *Proc. of PACT*, 2013.

[14] S. Ghosh *et al.*, "Cache miss equations: An analytical representation of cache misses," in *Proc. of ICS*, 1997.

[15] E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," *SIGARCH Comput. Archit. News*, 2008.

[16] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. of POPL*, 1988.

[17] M. K. Jeong *et al.*, "Balancing dram locality and parallelism in shared memory cmp systems," in *Proc. of HPCA*, 2012.

[18] M. Kandemir *et al.*, "Cache topology aware computation mapping for multicores," in *Proc. of PLDI*, 2010.

[19] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," *Proc. of HPCA*, 2010.

[20] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. of MICRO*, 2010.

[21] Y. Kim *et al.*, "A case for exploiting subarray-level parallelism (salp) in dram," in *Proc. of ISCA*, 2012.

[22] M. D. Lam *et al.*, "The cache performance and optimizations of blocked algorithms," in *Proc. of ASPLOS*, 1991.

[23] C. J. Lee *et al.*, "Improving memory bank-level parallelism in the presence of prefetching," in *Proc. of MICRO*, 2009.

[24] C. J. Lee *et al.*, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," *HPS Technical Report*, 2010.

[25] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Proc. of POPL*, 1997.

[26] J. Liu *et al.*, "On-chip cache hierarchy-aware tile scheduling for multicore machines," in *Proc. of CGO*, 2011.

[27] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 367–376. Available: http://doi.acm.org/10.1145/2370816.2370869

[28] Q. Lu *et al.*, "Data layout transformation for enhancing data locality on nuca chip multiprocessors," in *Proc. of PACT*, 2009.

[29] W. Mi *et al.*, "Software-hardware cooperative DRAM bank partitioning for chip multiprocessors," in *Proc. of NPC*, 2010.

[30] S. P. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proc. of MICRO*, 2011.

[31] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proc. of ISCA*, 2008.

[32] V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *Proc. of MICRO*, 1999.

[33] H. Park *et al.*, "Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," in *Proc. of ASPLOS*, 2013.

[34] S. Phadke and S. Narayanasamy, "Mlp aware heterogeneous memory system," in *Proc. of DATE*, 2011.

[35] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *Proc. of ISCA*, 2006.

[36] J. Ramanujam and P. Sadayappan, "Tiling multidimensional iteration spaces for multicomputers," 1992.

[37] S. Rixner, "Memory controller optimizations for web servers," in *Proc. of MICRO*, 2004.

[38] A. Sharifi *et al.*, "Addressing end-to-end memory access latency in noc-based multicores," in *Proc. of MICRO*, 2012.

[39] K. Sudan *et al.*, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *Proc. of ASPLOS*, 2010.

[40] I.-J. Sung *et al.*, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proc. of PACT*, 2010.

[41] A. N. Udipi *et al.*, "Rethinking dram design and organization for energy-constrained multi-cores," in *Proc. of ISCA*, 2010.

[42] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, 1991.

[43] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. of PLDI*, 1991.

[44] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, 1989.

[45] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in cmp systems via dynamic bank partitioning," in *The 20th annual IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '14, 2014.

[46] G. Yao *et al.*, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Syst.*, 2012.

[47] Z. Zhang *et al.*, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proc. of MICRO*, 2000.

[48] H. Zheng *et al.*, "Mini-rank: Adaptive dram architecture for improving memory power efficiency," in *Proc. of MICRO*, 2008.

[49] X. Zhou *et al.*, "Hierarchical overlapped tiling," in *Proc. of CGO*, 2012.

[50] Z. Zhu and Z. Zhang, "A performance comparison of dram memory system optimizations for smt processors," in *Proc. of HPCA*, 2005.